X00TIC *magazine*

June 2002-Volume 10-Number 1

POST-MASTERS PROGRAMME SOFTWARE TECHNOLOGY

Free the Software

Freedom

Making Money

An Insider’s View

Embedded Penguins

Contents

Colofon

XOOTIC MAGAZINE
Volume 10, Number 1
June 2002

Editors
C. Delnooz
N.H.L. Kuijpers
Y. Mazuryk

Address
XOOTIC and XOOTIC MAGAZINE
P.O. Box 6122
5600 MB Eindhoven
The Netherlands
xootic@win.tue.nl
http://www.win.tue.nl/xootic/

Secretariat OOTI
Mrs. C.I.T. Kolk-Koenraat
Post-masters Programme Software Technology
Eindhoven University of Technology, HG 6.57
P.O. Box 513
5600 MB Eindhoven
The Netherlands
tel. +31 40 2474334
fax. +31 40 2475895
ooti@win.tue.nl
http://wwwooti.win.tue.nl/

Printer
Offsetdrukkerij De Witte, Veldhoven

XOOTIC magazine
POST-MASTERS PROGRAMME SOFTWARE TECHNOLOGY

Free the Software

Editorial Preface

It’s been a while since the last issue of XOOTIC MAGAZINE appeared. A lot has happened since then. Around the time of publication of the last XOOTIC MAGAZINE, we celebrated our tenth anniversary with a symposium on Pervasive Computing. In February the board changed during the General Members Meeting. The hammer is now in the hands of our new chairman who leads a fresh board. Also the editorial board of XOOTIC MAGAZINE changed. Shortly after defending his Ph.D., Victor Bos left for Finland where he now works as postdoc. Consequently, he decided to stop editing the magazine. From this place I want to thank Victor once more for all his time and effort he has spent during the past years on editing articles and preparing the magazine for reproduction. Luckily, his vacancy was soon to be filled by Chris Delnooz who started the OOTI programme last September. The editorial board now consists of Yarema Mazuryk, Chris Delnooz and myself.

As a theme for this issue we have chosen *Free the Software*. We discovered that lots of XOOTICs are involved in the development or usage of free software. We thought that presenting their opinions and experience to the XOOTICcommunity might lead to an interesting magazine, and it sure did! We have four articles each focusing on another aspect of free software.

Andrew Mikheyev and Laurens Vrijnsen start off with an overview article in which they explain the concept of free software and present some successful open-source projects. They also make a comparison between open source and closed source on a number of criteria such as quality, usability, security, development speed and portability. The second article is written by Anthony Liekens who works as a Ph.D. student at the TUE. Although Anthony is not an (ex-)OOTI he is a regular visitor at the OOTI room. In his article, he discusses the business models that are applied for open source projects and he also mentions a few successful examples. XOOTIC member Koen Holtman writes about his own experience as the maintainer of the open source archiver AFIO. By focusing on just one project, he is able to describe the micro-process that open source developers are involved with. The fourth and final article is also written by an ex-OOTI: Ruud Derwig. Ruud writes about the increasing size of software embedded in consumer electronics equipment. He discusses how open source might be applied in the development of such software. He not only addresses technical aspects, but also organizational and, very important in this case, legal matters.

Enjoy reading this magazine!

Nico Kuijpers, editor

Advertorial: Philips

Page 4 (should be even)

Free Software = Software for Free?

Andrew Mikheyev and Laurens Vrijnsen

“Free as in ’free speech’, not as in ’free beer”’
(Free Software Foundation[2])

In this paper, we present the basic characteristics of free software and open source software. We illustrate their impact on today’s world of computing with a number of examples, before we compare the two against main-stream, proprietary software. The discussion will not be about holy versus evil, Redmond versus Red Hat, or Bill versus Linus; that has been covered by too many already. Rather, we will look at free software from a (system) developer’s point of view: how about quality, security, usability, development speed, portability and profitability?

Introduction

Free software, open source... just a few terms that are emerging these days. Many associate it with software hacked together by a group of enthusiastic amateurs from that ancient UNIX-world. Software placed on the Internet to make it available for everyone, for free, without guarantees. That can’t be serious software, can it? However... the growing popularity of Linux, also in the embedded world, is for a large part based on the fact that it is free software. But what does this concept mean?

In this article we will explore what “free” really means. First we introduce the concepts of free software and open source software. After a brief demonstration of its viability, we present a comparison of free software versus proprietary software. Finally we give some concluding observations.

Free software? Open source?

Contrary to what one may expect, the word “free” refers to freedom, not to “for free”. Free software offers following freedoms to its users:

1. The freedom to run the program, for any purpose;
2. The freedom to redistribute copies[1] of both source code and binaries so you can help others;
3. The freedom to study how the program works, and adapt it to your needs;
4. The freedom to improve the program, and release your improvements to the public, so that the whole community benefits.

A program is free software if users have all of these freedoms. Thus, you should be free to redistribute copies, either with or without modifications, either for free or charging a fee for distribution, to anyone, anywhere. Being free to do these things means (among other things) that you do not have to ask or pay for permission.

Open source is another commonly used term to refer to this type of software. A closer look at the two different movements “free software” and “open source” learns that they use almost the same criteria for judging software, but with different rationale:

- the free software movement has an ideological focus towards freedom for the user;

[1]An exception is made if export regulations are violated by this, e.g. for encryption software.

- the open source movement takes a more practical approach: it promotes software reliability and quality by supporting independent peer review and rapid evolution of source code.

In this article, we will focus on the practical aspects of free software. Therefore, we permit ourselves to use free software and open source software as synonyms. Instead of having developers that create software in their ivory tower and then give it to their customers, open source software (and free software) creates a community of developers and users that interact.

The above-mentioned freedoms have two intriguing consequences for producers of software:

- they may not get a fee for every copy that is used;
- their solutions to problems (as found in the software) are exposed to everyone outside of the company, including their competitors.

Many vendors of proprietary software use copyrights and patents to prevent users from claiming the above-mentioned freedoms. On a more practical note, these tactics prevent knowledge and ideas to spread and be improved by others, thus limiting the speed of development and progress in the field of computing science.

In spite of these consequences, more and more companies are turning to producing open source software, as will be demonstrated in the next section.

A few examples of open source projects

One of the most well-know open source projects is the Linux operating system. The open nature of its development has boosted its development and therefore has created the basis for its current popularity, both with “hobbyists” and professionals. People who have problems with their Linux find that the community is not only open for development, but also for providing fast and good support.

Apache, by far the world’s most popular web server with a 58% market share ([1]), gives another example of the high quality provided by open source software.

Apple was the first mainstream computer company to build its future around open source, and is partnering with the Apache Group, FreeBSD, NetBSD, and other open source developers to work on evolving the Mac OS X platform. It has released the core layers of Mac OS X Server as an open source BSD operating system called “Darwin”.

IBM chose the open source Apache web server to support and bundle with its WebSphere suite. It has since released the Secure Mailer in open source and launched a web site to distribute alpha-status IBM technology in source, before they are licensed or integrated into products. This allows developers all over the world to both evaluate and influence IBM research and development.

Sleepycat Software builds, distributes, and supports Berkeley DB, an open source embedded database system. Their customers include many of the leading open source projects, as well as Fortune 500 companies whose own products are proprietary.

These are only a few examples. We encourage the reader to search for open source products that match his/her personal preferences... there is a good chance you will be pleasantly surprised by the results.

As shown in the above examples, “free software” can be commercial. One has to realize that software is more than just a collection of bits: it is a product that requires support to tune it to specific requirements. It is beyond the scope of this introduction to elaborate on possible business models here; we refer interested readers to the article *Open Source Business Models* by Anthony Liekens, later in this magazine.

Open source versus closed source

So free software can be economically viable, but what are the benefits? In order to answer this question, we will compare open source to its opposite, closed source or proprietary software, on a number of criteria:

- Quality;
- Security;
- Usability;
- Development speed;
- Portability;
- Profitability.

Quality

The good quality of the final product is a sum of two major components: good design and good implementation. For a major part of open source projects we can say that usually both are at a very high level. Two factors contribute to this - the accessibility of the source code and the professional level of the developers.

The source code being publicly available can be analyzed by thousands of amateur-programmers or professionals whose interests lie in the field for which the product is being targeted. Everyone is free to update the source code or send a feedback to the author of the erroneous module in the case a bug is spotted. This tremendously accelerates the testing procedure of the product in comparison with the closed-source projects, where the testing is usually done by a limited number of beta-testers, and only the project's development team does the corrections in the code.

The availability of the source code partly explains the high quality of open source products. However, this is not the only reason for that. The professional level of the developers participating in open source projects is on average very high. The Boston Consulting Group in one of its surveys partially mentioned in [4] found that the open source developers surveyed are mostly experienced professionals having on average 11 years of programming experience and the average age of 28.

Peer reviews play an important role in the open source development process and contribute to the high quality of the resulting products as well. Since all open source developers can see source code produced by the others, they can spot defects in the code and provide its author with feedback. If low quality of the source code becomes a persistent issue for some developer, then eventually he will have to leave the community.

All these factors contribute to the quality of the open source products, allowing them to score better in this category than the closed-source products.

Usability

As mentioned before, almost all open source projects are carried out by people who are fluent in modern software and hardware technologies. Traditionally, those people tend to concentrate more on the technical side of their work rather than paying attention to such details like user interface design. The implementation of a convenient user interaction in their products is not at the top of their priority lists. This is where the commercial closed-source products (usually working under Windows), definitely beat open source products with their amateur-like user interfaces.

Meanwhile, the open source community seems to have finally understood the problem. The situation with the usability of the open source software is constantly ameliorating. For already several years, newer versions of popular desktop managers for Linux having a constantly improving user interface are a good example of this positive trend. However, the developers still seem not to have found the right balance between the amount of functionality they offer in their interfaces and their ease of use.

Security

In the software systems there are many ways how security holes can appear. They can be caused by a bug that makes the system behave in a non-specified way. They can also appear as a side effect of some feature of the system - of which no one had ever thought before. The communication protocols used or implemented by the system can be poorly specified and use of them in an improper way can lead to security problems as well.

What will developers do in order to spot all the potential sources of security problems?

In the closed-source world testing is performed inside the company where the product is being developed. Some companies even hire professional hackers and let them explore the source code and the product itself to find as many potential security issues as possible.

In open source, all software developers of the world can have access to the source code of the open source products. If someone suddenly discovers a security problem, it will be known very soon by the open source community and the necessary measures will be taken by the authors of the system or concerned users. The fact that the source code of all widely used products is being constantly analyzed by thousands of software specialists all over the world raises the security of those products to the level yet unreachable by the closed-source software industry.

Another advantage of open source is that using an open source product you can be sure that it doesn't contain any sort of back-doors - a hidden functionality that can be activated and used by the author of the system, intelligence or military organizations - without keeping you aware of this. As long as you have the source code of the system, it will be impossible to hide anything like this inside of it.

Development speed

The open source projects are usually developed by teams consisting of many people distributed all over the world. Most of them works on the project during their spare time, taking no obligations of any kind before the community. Some people do it because they believe source code should be open, others participate to improve their programming skills or just for fun. There are also people who do it for their professional needs, working on the parts that they need themselves. In all cases, the level of motivation of the developers is high enough to compete in development speed with the commercial closed-source projects.

Since Linux appeared in 1991, its today's releases contain tens of millions lines of code - all written by the participants taking no obligations of any kind before the community. Thus, Red Hat Linux 6.2 contained over 17 million lines of code, and Red Hat Linux 7.1 is composed of 30 million lines of code which is even more than those 29 million lines of Windows XP, which is considered to be the largest commercial project ever carried out! These figures are not only a testimonial of the high development rate that can be reached in the open source projects, but these figures also give us an evidence of a very high potential scalability of the open source development process.

However, open source development strategy has its drawbacks. The non- obligatory participation in the projects makes it possible for every participant to stop contributing whenever he wishes so. As a consequence, it is almost impossible to predict the release date for a next version of any open source product.

Another disadvantage of the open source development process is its development latency for supporting new hardware. One can run into troubles trying to install Linux on a brand new machine equipped with the latest graphics card, wireless connection card and other just released hardware equipment due to the lack of drivers for all this hardware.

Portability

Portability is becoming a very important concern for the developers who are working on the non-PC-based platforms. Embedded systems developers, for example, would greatly benefit from the possibility to tailor an external piece of software for their own hardware configuration. This is where open source solutions are much more attractive than the ones using closed-source ideology.

At present moment, many companies are working on their own versions of Linux for use in their proprietary embedded systems. This dispenses them from developing new operating system from scratch.

NetBSD operating system is just another good example of the portability of open source solutions. Up till now this operating system has been ported to as many as 48 different platforms! Different development teams got the possibility to port NetBSD to the platforms they are interested in, since its architecture and source code are publicly available for downloading.

Such an activity wouldn't be possible if the source code of the system had been proprietary and closed. The company-owner simply wouldn't have coped with the task of porting the system for so many hardware platforms. Most likely, it would favor one hardware configuration (one specific CPU) and produce builds for this particular device. This kind of strategy has been undertaken by Microsoft with their latest PocketPC 2002 operating system for which it had been announced that only Intel's StrongARM processors would be supported starting from that version.

Conclusions: applicability?

So strangely enough, free software seems to be most appropriate for those who are willing to pay for it. In the market of embedded software, it can lead to closer ties through co-development. Instead of selling software, companies can focus on selling support, e.g. tailoring software products to unique customer requirements. Open source software allows for fast progress in development of new software products by sharing new ideas. Exactly this is the secret to why free software products outdistance their commercial counterparts on a number of aspects.

For the large group of home users and office automation, open source software is becoming more and more attractive as an alternative for expensive software products. However, how a company can sell support to this group of customers remains unclear. Therefore, the viability of delivering open source products to this group is questionable, but companies must react to the competition offered by high-quality open source software products.

References

[1] Netcraft, ”Netcraft Web Server Survey”, http://www.netcraft.com/survey/

[2] Free Software Foundation, ”Philosophy of the GNU Project”, http://www.gnu.org/philosophy/

[3] The Open Source Initiative, http://www.opensource.org/

[4] Why Open Source Software? Look At The Numbers!, http://www.dwheeler.com/oss_fs_why.html/

Andrey Mikheyev holds an M.Sc. degree in mechanics and automated control received from French Graduate School of Mechanics and Microtechniques (Besanon, France). He also received an M.Sc. degree in computer science from State Power Engineering University (Ivanovo, Russia). He is an OOTI trainee since September 2001.

At this moment, apart from other professional interests, he is particularly interested in all products and technologies offered by Microsoft since he thinks that this passion will help him answer the ultimate question - “What does an IT-company and its employees need, to develop great products and thus survive and flourish on the today’s hi-tech market?”

After a nine-month research project in Philips Research on a software architecture for the domain of emergency medical care, **Laurens Vrijnsen** received his Masters degree in Computer Science from the Eindhoven University of Technology in August 2001. Shortly afterwards he joined the OOTIprogram. His current fields of interest are software architecture methodologies and autonomous systems.

Laurens’ experience with UNIX and free software dates from 1997, when he was introduced with the FreeBSD operating system. Ever since he has been a devoted worshiper of daemons and the UNIX design philosophy: creating small, reliable solutions.

Advertorial: Thales

Page 10 (should be even)

Open Source Business Models

Anthony Liekens

At first hand, Open Source Software (OSS) and conventional business models do not seem to match. OSS often gets the connotation of being free of charge, which is not an encouraging prejudice when one is trying to make money out of it. However, OSS can offer some new opportunities for commercial software development.

How can a firm in the sector of information technology make money out of Open Source Software, while supporting the community that thrives on free software? To start with, an entrepreneur who is developing OSS, is not simply supporting the open source community. OSS is a way of building software in collaboration with the users of the software packages, which can possibly end up in creating a product with a level of quality that could not be achieved with closed source. The user is given the ability to propose useful bug fixes and interface changes. This close interaction with the user can obviously lead to an improved product, and in many examples on the Internet, open source development is starting to beat the monopoly of commercial closed source software farms.

Open source development gives customers a much greater ability to customize software to fit their needs. Customer bug fixes and enhancements are commonly contributed back to standard open source packages, while improving upon the quality and limitations of the product. This option is not available within traditional commercial software.

Do you trust your toaster's software?

One of the most discussed results of this interaction is the increased security and reliability of OSS products. Giving away the source code of a software product will give the community of users the possibility to expose the product to increased testing, such that security problems and fixes can be discovered and distributed earlier than in a closed source model, improving the overall quality and reliability of the software package.

Entrepreneurs using OSS business models depend largely on the high reliability of their software; their software is peer-reviewed, and possibly tested more extensively than proprietary closed source software. Opening the source of a project can make the project free of bugs, and as bullet-proof as software can get when reaching a mature stage in the software's life cycle.

Advantages of OSS

Speeding up development

Allowing a product's users to be co-developers along with the product's developers seems to be in stride with conventional ideas of software development. Indeed, OSS development is based on the idea that more programmers can accomplish more than the selected few in a company's developer room. It just follows along the lines that more eyes simply see more. The more programmers are poking a project's source code, the faster bugs or security flaws can be detected, and the faster the development process can lead to a mature product.

This interaction with the users is, however, not as easy as stated here. Users are not always happy to pay for an unfinished product. It is getting even worse when they start using it, discover flaws in the program, and then have to put energy in further development of the product.

The manufacturer of the product also has to provide an initial full version of the product to receive any interaction with other developers. From this first version, the entrepreneur can start attracting users and their developers, after which the speed in development and the evolution of the product could increase, given that the initial version satisfies the costumer such that he is willing to put money in the further development of the product.

Lower overhead

External co-developers, given the opportunity to resolve bugs, can be adopted to out source part of the work of a company's software shop. In return of fixing a bug, that costumer can be given an opportunity gain over other costumers. This allows to reduce per-project software production costs significantly. And as an extra, a small developer team can handle a much bigger project.

Closeness to costumers

It is very favorable for a company to be close to its customers. In the case of a company providing software, there is no better way than allowing your customer's engineers to be involved in the software project's development. Their involvement in the development of the source code allows them to easily fix the flaws that limits their productivity, again allowing for a better product in the end.

Open source gives customers a much greater ability to tailor software to fit their business needs. Customer bug fixes and enhancements are commonly contributed back to standard open-source packages, an option which is not available with traditional commercial software.

Broader market

Allowing a customer's engineer to be able to adapt a product, allows this user to extend the product beyond the limits of what a company originally intended with the project. If these adaptations are returned and merged with the original product, it can attract more customers to the product. As an example of this, a customer might port the code to a new platform giving him, and possibly other customers, the ability to use the product in their working environment, beyond the initial limitations implemented by the manufacturer of the product.

Public relations

Giving away (limited versions of) source code and products for free allows new users to test and compare your product to other products, which again can attract more customers to start using the full product.

Making money out of OSS

Once a software manufacturer uses OSS to create software, it is not always clear how he can make money out of the project. There are a few relatively new business models which adapt open source development and offer opportunities to make open source development worthwhile.

Supplying service and support

If a software project is distributed for free, the users of this project might not always be able to use the project to its full extent. In a commercial environment, the user can pay for service and support of the project. Even the implementation of a complex open source project could be out sourced to the creator of the original project. A good example of this business model is used by MySQL AB, which is discussed later.

Loss leader market entry

The loss leader OSS business model is often used for two purposes. Firstly, it can be used for jump starting an infant market, and secondly, it can be used to break into a market with entrenched closed source players. Many funding in open source projects can be viewed as strategic loss leader models against popular, possibly monopolizing closed source software companies. These investments are best done at the steepest part of the product's growing curve. A good example of a company using the loss leader model, is Netscape, which opened up the source code of its Netscape Communicator web browser to attract developers and users to open up the market of Internet browsers currently monopolized by Microsoft's Internet Explorer. Netscape is also a good example of how users can adapt the project and create a much improved product, Mozilla in this case, from the free source.

Widget frosting

Many hardware manufacturers have to provide software – such as drivers or other interfacing software – along with their products. Using the open source model in the development, along with opening up the standards and technologies used in the hardware, allows the company and users to create software that works on platforms and with ideas beyond the limits the manufacturer originally intended. The production and extensibility of the software offered along with the hardware sold, might attract more buyers of the product. An obvious example of this model is that manufacturers of graphics cards for personal computers can attract more buyers because their open sourced drivers are ported to new platforms the manufacturer originally did not envision.

Accessorizing

Companies such as O'Reilly Associates, SSC and VA Research base their success on selling accessories based on open source projects available to anyone. They offer books, compatible hardware or complete pre-installed systems based on open source software. Since the software they build upon is available for free, pre-installed systems can be built with a very low cost on licensing, and the final product can stand out against systems built upon commercial software.

Examples of successful business models

A couple of successful companies, who base their business on OSS are illustrated in the remaining part of this article.

Redhat network

Redhat, among many other companies such as Suse or Mandrake Linux, is a provider of distributions of the open source Linux operating system. Specifically, Redhat network offers services to easily maintain Linux installations. The company offers service, support and training for administrators of Redhat Linux installations. A registered user can obtain updates of the operating system, and call a help desk if technical problems are encountered. On the other hand, the company also provides consulting services, such as high performance computing or web services, all based on free open source software. To satisfy their users, they have developer teams working on user interfaces and enhancements of the open source software they adopt, while supporting the community of open source developers.

MySQL AB

The German company MySQL AB originated from a group of developers who created the open source database SQL server MySQL. The company offers service solutions and training based on the free software product. The biggest part of their revenue is obtained from professional consultation for the implementations of their free product in commercial environments.

Ximian express

Ximian provides desktop solutions for the Linux operating software. Everyone is free to download their product, but a subscription allows the buyer to have priority access and higher bandwidth Internet downloads of the updates of their products.

SourceForge enterprise edition

VA Research is offering an open source version of their SourceForge product as a free software package to manage software development. Next to offering this version for free, they extend the package with other services, and the whole is sold as an enterprise edition of the free product. This enterprise edition contains extra enhancements and functionality which is not available in the core product that is offered for free. This allows the company to create a big user base, and rock solid product through open source development, to attract commercial buyers for its full product.

Others

This list can be extended with many other companies supporting and adopting the open source model. Among these are for example IBM who currently starts shipping Linux based systems, or SGI who is supporting the development of Samba, a communication interface between Unix and Windows system, while selling a commercial version of the package for its IRIX operating software users.

Discussion

Contrary to what is thought of open source as free-of-charge-software, there exist a couple of opportunities to adopt open source software in commercial environments. The advantages, however, are not always applicable to every software product, but many tricks can be used to create a software component that can be adopted both in an open source and commercial product environment, while inheriting advantages of both worlds.

Anthony Liekens studied computer science and biology at the Vrije Universiteit Brussel and at the Universitair Instituut Antwerpen, both in Belgium. He is now engaged in a PhD program in Biomedical Informatics at the faculty of Biomedical Engineering at the Technische Universiteit Eindhoven. His main research is situated in population genetics and genetic algorithms. Besides his work, he is involved in several open source projects. Anthony is completely incompetent in using Microsoft products, but fits in well when seated at open source powered machines.

AFIO: Inside an open source project

Koen Holtman

What do people actually do when they work in an open source project? What is the software process? Below I try to answer such questions by describing one particular case: my own work on afio, an open source archiver program that was initially created in 1985, and for which I have been the maintainer since 1993.

Introduction

While a lot has been written about open source software, much less has been written about the process of creating open source software. I know of a few good general accounts, which I refer to at the end of this article. In this article I will not describe the ‘typical’ or ‘average’ open source software process. What is average is a difficult question anyway, and depends in part on how broadly you define open source. Here, I give an account of my own activities in doing open source. I focus in particular on the case of the afio program. By using a specific case I can describe details of the micro-process that are often not covered in the more general accounts of open source development.

The afio archiver

Afio is a Linux/Unix program for packing up files into archives, and writing these archives to devices. It is very similar in function to the Unix ‘tar’ and ‘cpio’ commands, and the pkzip package in MS-DOS/Windows. Figure 1 shows the ‘official’ short description of afio that I bundle with releases.

Archiver & backup program with builtin compression Afio makes cpio-format archives. Afio can make compressed archives that are much safer than compressed tar or cpio archives. Afio is best used as an ‘archive engine’ in a backup script.

Figure 1: The short description of afio in its Linux Software Map entry

The main attraction of afio, over the better-known tar, is that afio makes compressed archives in a safer way: it compresses the individual files in the archive, rather than the complete archive byte stream like tar does. If a compressed tar archive encounters even a single byte error on reading, the remainder of the archive cannot be unpacked anymore, and all the data in it is lost. Afio archives are more fault tolerant: a read error will generally only affect the unpacking of a single file.

Early history of afio

The first version of afio was written by Mark Brukhartz in 1985, or possibly even a few years earlier. I never talked to Mark Brukhartz, so I do not know exactly why he started afio. My guess, from the documentation he included, is that he needed a better version of the cpio program which he was using, and that he did not have the cpio source. In terms of software years, afio is ancient, and that is part of the attraction in maintaining it. They really wrote C differently back then. This is a typical fragment of the 1985 afio code, which is still part of the source today.

```
/*
 * inavail()
 *
 * Index available input data within the
 * buffer. Stores a pointer to the data
 * and its length in given locations.
 * Returns zero with valid data, -1 if
 * unreadable portions were replaced with
 * nulls.
 */
STATIC int
inavail (bufp, lenp)
     reg char **bufp;
     uint *lenp;
{
  reg uint have;
  reg int corrupt = 0;

  while ((have = bufend - bufidx) == 0)
    corrupt |= infill ();
  *bufp = bufidx;
  *lenp = have;
  return (corrupt);
}
```

In 1985, Mark Brukhartz added an open-source type license at the start of the afio source, and distributed it to others in the Unix community. I don't know exactly what distribution mechanism he used, but it was not FTP on the early Internet. In 1991, someone called Jeff Buhrt added the fault tolerant compression feature to afio, and distributed the improved version, probably by posting it to a Usenet newsgroup. In any case, soon afterwards an Englishman called Dave Gymer downloaded afio from Usenet and started using it. In 1993 he made a Linux port, and uploaded it to sunsite.unc.edu, then the major FTP site for Linux application software.

How does one get involved in an open source project?

At one point in early 1993, I had a bad experience with the fault tolerance of tar, so I went looking on the Linux FTP sites for a more fault tolerant program. Afio was the most fault tolerant program I could find. Afio did not have all the types of fault tolerance I wanted, so I started writing my own backup program called tbackup which would use afio as a main component. The main added feature of tbackup was the fault tolerant and user friendly handling of cheap floppies as a backup medium – I had many boxes of cheap floppies lying around, containing outdated versions of the Slackware Linux distribution.

Pretty soon I found that afio would crash in compressing large files if the hard disk was nearly full. So I changed the code to fix that, and also mailed the changes to Dave Gymer, for him to incorporate in new afio releases. Fixing the code and mailing the fixes were fairly natural things for me to do. From my late 1980s home computer hobbyist days I was used to writing my own improvements to other people's code. Also I had already been using BBS systems and Internet mail for some time, so I was used to communicating over the net with complete strangers. It was obvious that the Linux community was just another bunch of computer hobbyists working in a gift economy, similar to my old home computer club. All things considered, it required no big conceptual leap for me to become a Linux open source contributor. It was just a combination of behavioral patterns and rules that I knew already.

At one point, in an e-mail exchange with Dave, we came to the joint conclusion that I was making more frequent changes to the afio code than he was. So we agreed that I should make the future afio releases. In December 1993 I uploaded a new version of afio, version '2.3.5 for Linux', to the usual Linux FTP sites. I had updated the documentation so that future bug reports would be sent to me. I was now the official maintainer of afio for Linux. People with other Unix versions also picked up the new afio for Linux version, ported it to their systems, and sent me portability patches. So after a few versions I dropped the 'for Linux' from the version designation.

What does an open source maintainer do?

Here is the process that I use to maintain afio. It has remained more or less the same over the years, even though, since 1993, the size and composition of the Linux community has changed drastically. I did not get this process from a textbook, nor did I first study other open source efforts to see how they did it. I started doing it this way because it seemed to be the obvious way to do things. (It is actually an interesting question if the open-source community is self-selecting for personality types to whom a certain way of working is the obvious way to work. Reading media accounts of how other people in open source do things, what strikes me most is that I find everything completely obvious, while the journalist writing it often expresses wonder at how things are done.)

Getting e-mail about afio

A big part of the maintenance process is dealing with the e-mail I get about afio. I get on average about 5 new mails related to afio each month. I can answer about half of these mails with a single reply, the rest lead to a series of message exchanges. On average, handling the mail takes me about 10 hours per month. I am very careful in archiving all the mail, to make sure that I will account for all contributions and bug reports when making the next release.

I try to reply to every new message within a week – if I have no time in that week to address the message I just send back a short reply that I am very busy and that I will give a full response hopefully within N weeks. I consciously work to give the impression that something will really happen with the mail people send me. The last thing I want is to discourage people from sending me more contributions in future. Of course nothing bad will happen to me when my correspondents get unhappy about the way I treat their mail. But I am working from the principle that everything worth doing is worth doing well. As long as I choose to fulfill this role as a maintainer, I want to keep up the same standards of service that I would like to see in any other software project, be it commercial or open source.

I use an informal tone in my replies to e-mails, but I consciously try to be polite and clear, even if I think that the original question is stupid or wastes my time. I actually get very few stupid questions, and most of the mail I get comes, as far as I can determine, from people who are already somewhat experienced as a Linux or Unix system administrator. In the last year I have started to get some mail from commercial Linux system support companies, who are asking me about problems reported by their customers. Again I treat these the same as any other mail.

Mail with questions

About half of the e-mail I get is some kind of question: how can I do X with afio? What does this error message mean? Is it available on platform Y? I can usually address these questions with a single reply. Sometimes I can only give a preliminary diagnosis and have to ask for clarifications or more information. In about half of the cases, if I ask for a clarification of a vague question, I do not get any reply. Presumably the person asking solved the problem already. About half of the questions I get are answered in some way by the manual page or release notes. I don't know the complete manual page by hart, so I usually have to look myself to see if the answer is in the documentation – if I find it I summarize what to do and then often cut-and-paste from the manual page in the reply. About half the questions I get uncover some weakness in the documentation, which I then often fix in the next release.

Mail with bug and problem reports

A second class of e-mail, about one fourth of the total volume, reports some afio behavior that was not expected by the user. The message I get can be a detailed bug report, but most often I get a cut-and-paste of the afio command line used, some error messages, and a partial description of the system configuration on which afio was run. Sometimes the observed behavior is actually correct according to the manual, and the user just expected something different. More often the behavior is something that should really not happen. I always end replies to such messages with some variant of 'thank you for this bug report'.

Sometimes I have seen a similar problem reported before, and I can search back in my mail archives and software change log to find a reply. If the problem is new, I try to reproduce it on my own machine. I manage to reproduce it about one third of the time. If I cannot reproduce it I will ask for more information, or for the results of some specific tests. In the end, about one third of the reported problems remains unresolved – sometimes with suspicions that the real source of the problem was a bug in the device driver of the backup device, but often with none of us having a real clue about what went wrong. Leaving something unresolved is frustrating, but at one point I have to decide to stop trying. Often, my correspondents are happy anyway when I tell them that I have given up, because in additional tests they ran the problem never occurred again.

Mail with contributions

A third class of e-mail, about one fifth of the total, contains a contribution to the afio code or documentation. Contributions to the documentation are fairly rare. I usually get code, in the form of patch files. About one third of the code contributions are bug fixes, about one third compatibility fixes to port afio to a non-Linux platform, and about one third are new features. When I get code for a new feature, it rarely includes any documentation that is good enough to paste directly into the manual page. I never ask contributers to write the missing documentation, I just write it myself. Many of them are not native speakers of English anyway, and I would not want to annoy them by drastically re-writing their attempts before inclusion.

In a few cases I reject code for a new feature, saying that I will not fold it into the release. This is either because I believe that the function can already be achieved in another way, or because I believe that the feature is just a bad idea, that would take too much of my own time to fold in and test. However, overall I hardly ever reject anything, and as a result the number of command line options to afio has grown from 36 in 1993 to 60 now, using all lower case letters, all upper case letters, and most of the numbers as option flags.

I always make sure that I give feedback to code submissions, either with a statement that I won't put the code in, or more usually with a statement that 'I will probably add it to the next release, which will come out in [time estimate in months]'. I write that I will 'probably add', because at the time of replying I have not yet made a full evaluation of the contributed code. I only take a very close look at the code when I start to prepare the next release.

Other mail

I also get a few e-mails which do not fit any of the above categories. Very rarely I get a 'thanks for afio' message without any further questions or requests. Very rarely, I get a plain request for a new feature. Sometimes this feature is something that can already be done with afio: if so then I write back how to do it. If it cannot be done yet, I generally comment on whether and how it could be implemented, and encourage the requester to send in an implementation. Usually I do not get any. Sometimes, if I believe that the idea for the feature is a good one anyway, I implement it myself at the time of the next release.

Making a release

I do not release new afio versions often. In the last few years, the release frequency has been about once every 9 months. A few releases were prompted by the discovery of critical bugs that should be fixed urgently, but usually I release when I have a sufficiently large number of patches and bug reports, and when I can find the time to fold them all in. Making a new release costs me about 40 working hours: I work in evenings and weekends over a period of a few weeks. During that time, I re-visit all archived mail since the previous release, changing the afio code and documenting the changes as I go along. At the end I do regression testing, create a new source archive, and upload it.

Afio is a mature backup program that people rely on. My first order of business is not to introduce any additional bugs, and this drives my release strategy. For other open source programs, which are early in their development lifecycle, the strategy is to release very often, relying on the early adopters to find the bugs in the new code. With afio I also rely on users to find bugs, and this user testing adds significant value, but the type of bugs people find are the very obscure bugs that are left in a well-aged program. For example, a recent bug report had to do with the incorrect handling of Unix file system symbolic links that have several hard links to them. Other bugs that people find in afio are those triggered by new use cases. Over the years, as hardware capacity grew, I first got bug reports related to making multi-tape archives larger than 2 GB (2^{31} bytes), then about handling tapes that are individually larger than 2 GB, then about archiving single files that are larger than 2 GB.

The release process

When making a new release, I carefully hand-check and test all code contributions. Often I make substantial changes to contributed code to make it more safe or general. The new idea behind the code, or the finding of the bug that the code fixes, is often the most valuable part of a code contribution. Reviewing and testing new code takes significant work, but it is needed to maintain the most important feature of afio: its stability. This careful review of all code is not unique to afio: I have also seen it in the maintenance of the Apache HTTP server, another mature piece of open source software.

Coding is actually a very small part of making a release. I spend most of my time testing and updating the documentation. When I add a new feature, I often spend more time updating the manual page than updating the source code – writing a terse but complete description of a new feature and its limitations is surprisingly hard. I also spend significant time updating the change log file that is bundled with the afio sources – see figure 2 for a typical excerpt.

Version 2.4.7:

Fixed bug that sometimes caused '– compressed' to be printed twice in verify operation. Has to do with not flushing stdout, stderr before forking. Bug reported by JP Vossen.

Added more material on how pattern matching works in the -y option section of the manpage, and added examples of selective restores to manpage. Based on questions by Kjell Palmius and Stojan Rancic.

Added text to BUGS section about afio not being able to write into directories for which it has no write permissions, except when running as root. Problem reported by Kagan Kayal.

Figure 2: A part of the afio change log

The change log has two main functions. First, it helps me and other contributers to keep track of changes and solved problems. Having a very detailed change log saves me significant time in answering e-mails. The second function of the change log is to record the names of all contributers to afio, which I define as everybody who sent me any mail that led to changes in the next release. The change log gives visible evidence that even the smallest contributions are welcome, and will have an effect. I have a theory that this is very important in encouraging future contributions. I never record the e-mail addresses of contributers in the change log, because automatic publication of their e-mail address might actually discourage people from sending me mail.

The last part of the release process is to make a new source archive, upload it to the various repositories, and write an announce message for the comp.os.linux.announce newsgroup. This always takes a surprising amount of time, because you do not want to make any last-minute mistakes. I need to spend some time for every new release to catch up with recent changes in publishing Linux software. Back in 1993 it was sources on FTP sites and a message on an announce newsgroup. Now it is mostly web sites and pre-packaged pre-compiled binaries. However, I still don't build pre-packaged binaries, and I do not get deeply into the web site stuff: I have decided that I'd rather spend my time on other things. Because I don't make pre-compiled binaries, afio probably has less users than it could have. But that is fine with me – I am not in this for world domination, and have no obligation to spend my time serving all Linux users optimally. My main interest is to give to other programmers in the open source community, who will be served about as well with a source release. Somebody else, in the Debian Linux distribution effort, does in fact maintain a Debian binary release of afio, and we have very friendly relations. When I am about to make a new source release, I give him an advance warning. When he gets a bug report about the binary release that has to do with the source, he copies it to me.

What is the motivation for doing it?

Software maintenance is perhaps not a very obvious thing to do in your spare time. My reasons for doing it are partly historical. I started out mainly as an open source author – I like writing software and if I do it in my spare time I like to share it with others. Over the years, I found that I had less and less spare time which I wanted to devote to programming. So I stopped writing new code and only did maintenance on the old code in the projects that I happened to run. By 1995 I was doing maintenance on 3 open source projects: afio, tbackup, and futplex. Still I had less and less time: I found that my release frequency was dropping to below what I found reasonable. So after 1995, I first stopped maintaining futplex, which had never attracted a very active user community anyway. Later still, I also stopped maintaining tbackup, which did have a user community, but one that generated much less feedback than the users of afio. I still get messages about tbackup, about one every two months. I always reply that the software is now 'dead' and unsupported, and that I recommend using another backup program even if tbackup does still install.

Maintaining afio is not very creative work, though there is occasionally an interesting puzzle. Doing the maintenance process is mainly rewarding, I guess, in the same way that gardening is rewarding: there is usually no great pressure, you get to do something immediately visible with your own hands, and it is nice to make things more tidy. I also find it rewarding to help complete strangers who e-mail me. I am a heavy user of open source software myself, and it feels good to be in a position where I am not just taking, but also giving back to the open source community. Occasionally, it is fun to consider that I have achieved some degree of immortality through my work, because the code I wrote has been pressed on lots of CD-ROMs. However this is more of a fringe benefit, it is not a state of mind that I can sustain indefinitely.

Starting an open source project

Starting an open source project is easy. Create some software (but make sure that you do not incorporate any third party code that is restricted or proprietary), bundle the sources together with a file that identifies you as the author/maintaner, and upload the result to whatever the usual places are for your platform. That is really all there is to it. A web page is optional, but very much expected these days.

Of course there is no guarantee that a new project will generate the level of interest that you expect. In my own new projects, I never tried to predict or optimize the level of interest beforehand. When my code was at a stage where I thought it to be potentially useful to others, I just wrote the amount of documentation that I judged to be necessary to support other users and developers, bundled it, and made the first public release.

Though I did not try to guess levels of interest beforehand, the levels that I observe during a project do influence my actions, in particular when choosing which projects to stop.

Further reading

In recent years, the new economy has become somewhat of a spectator sport, generating a matching volume of mass media accounts and Internet discussions. Microsoft litigation and open source are particularly popular subtopics. On the Internet, participating in discussions about open source has become a legitimate full-time hobby all by itself. Most of the written material can be safely ignored, unless you happen to be a fan.

The Cathedral and the Bazaar by Eric Raymond [1] is a classic and thought-provoking essay that contains some good descriptions of the open source process. The essay is thought-provoking because it argues, with some actual proof, that very complex software projects can be run successfully, even optimally, without much central planning. I don't believe all claims of the essay, but that is exactly what makes it thought-provoking. Don't miss the [EGCS] end note which is present in the more recent releases of this essay. There are several followup essays by the same author, but I find these less thought-provoking.

Hackers by Steven Levy [2] is a well-written book, written in 1984, about several open source cultures that pre-date Linux, about their development processes, and about their interactions with the market economy.

A recent statistical survey of open source authors is at [3]. Like many statistical surveys, it offers few real surprises, but it does solidly contradict the view that open source authors are an untrained mob of young computer nerds out to destroy Microsoft.

References

[1] http://www.tuxedo.org/˜esr/writings/cathedral-bazaar/cathedral-bazaar/

[2] Hackers, by Steven Levy. 1984. http://mosaic.echonyc.com/˜steven/hackers.html

[3] http://www.osdn.com/bcg/

Koen Holtman is a researcher at the California Institute of Technology, where he does architecture and coordination work in developing the world-wide distributed data processing infrastructure of the CMS particle physics experiment. The current version of this infrastructure relies heavily on Linux systems, and several of the development partners publish their middleware with an open source license.

Koen was an OOTI from 1995-2000, finishing with a Ph.D. in software design for work performed at CERN in Geneva, Switzerland. He also holds an engineering degree in computing science from the Eindhoven University of Technology. He has been active in the Linux open source community since 1993, and also contributed code to the Apache server project.

Advertorial: Sioux

Page 22 (should be even)

Linux inside your TV?

Ruud Derwig

The Internet has enabled 'open-source' software development, a process that uses independent peer review and rapid evolution of source code to improve reliability and quality. Open-source licenses allow software modification, (re)use, and redistribution, giving users perpetual, full access to software, independence, choice, and control. This 'collective' ownership motivates people to contribute to the code base. The GNU/Linux operating system is an example of a successful product of the open source community. During the past years Linux has gained a strong position in the (Internet) server domain. Whether Linux and other open-source software is an option for consumer electronics products like television sets, DVD players and set-top boxes is the topic of this article.

Introduction

Consumer electronics equipment found in people's homes today contains more and more software. The functionality and complexity of home products increase steadily and the size of the software needed to realize these products is growing exponentially. Whereas a few years ago the sizes of embedded TV and set-top box software were measured in tens or hundreds of Kbytes, today's high-end products contain Mbytes or tens of Mbytes. And all these Mbytes have to be filled by (expensive) programmers. Wouldn't it be nice to obtain large parts of that software from third parties, and wouldn't it be even more nice to get it for free? This promise - and a dose of Microsoft antipathy - is what lead to many designers and companies in the consumer domain taking an interest in Linux and open-source software. In order to discuss the possibilities of developing consumer products with Linux inside, first we present a number of trends and associated features that will be introduced in new products the coming years. Furthermore, some of the important requirements of the high-volume electronics domain are explained. Then we delve into a number of technical issues that are relevant for deciding on the feasibility of Linux inside your TV or other consumer product. Next to the technical issues, however, it might turn out that non-technical issues like support, licenses, and development process are even more important. After discussing these non-technical issues this article is concluded with the answer to the question: Linux inside your TV?

Tomorrow's products

Trends

The Internet is entering the living room. Although not spreading as fast as people thought - or hoped - some years ago, new features that require Internet connectivity are being introduced right now. Many commercials being broadcast at Dutch television already present a URL pointing to a web page with more information about a certain product. Instead of running to the study, booting your PC, trying to remember the URL, typing it into your browser and finally getting to the content, wouldn't it be nice to push a button on your remote control and get the information on your TV screen? Several digital TV service providers already have a web browser integrated into their high-end digital receivers. But web browsing on TV is not the only feature that needs an

Internet connection. A network connection can also be used for getting data that is not web-related like e.g. the information needed for an electronic program guide. Or it can be used for streaming audio or video content, like for instance in an audio set supporting not only traditional AM and FM bands, but also IM - Internet Modulation - for receiving Internet Radio stations.

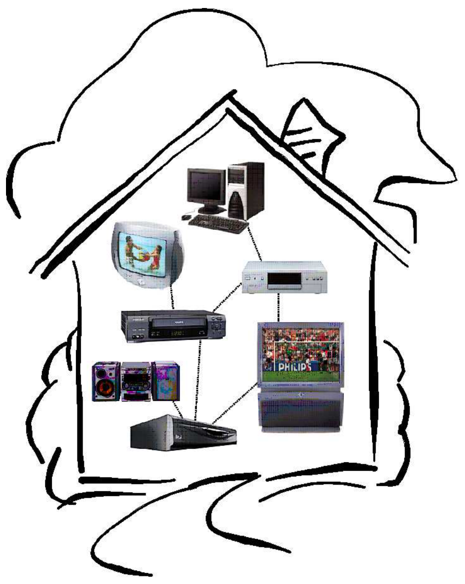


Figure 1: Connected consumer devices form in-home networks.

The Internet is not the only network entering the home. Whereas today devices are operating stand alone, with at most a cable connecting the VCR or DVD player to a TV, in the future devices in the home will be connected - either wired or wireless - to form in-home networks. Instead of always having to watch satellite channel movies in the living room, because there the set-top box is connected to the TV, the set-top box could redirect the movie through the home network to the bedroom and the bedroom TV could forward remote control commands back to the set-top box. The same scenario applies to other devices like for instance a storage device. While watching the forwarded movie in the bedroom, for instance, you get too sleepy to continue watching. Instead of getting out of bed, sleepwalk to the VCR and program it, a storage device connected to the in-home network could be controlled from any place in the house. Although a fully connected home network is probably not going to be there in the near future, the first digital receivers supporting a second TV will enter the market soon.

Another development that is important for the Linux discussion is the (r)evolution of hard-disk technology. Storage capacity of hard-disks is growing exponentially, even faster than Moore's law. Using compression, it is possible to store about an hour of high quality video in a Gbyte. This means that a 60 Gbyte hard-disk can store 60 hours of video. Because hard-disk bandwidth is high enough to support several video streams simultaneously, a hard-disk enables new features like a pause button for live broadcasts. The first hard-disk based video recorders are available today.

The last trend we mention is the fact that consumer systems are becoming more open, like PC platforms. This means that new, third party applications can be downloaded and executed on existing products. An example from the digital video broadcasting domain is the Multimedia Home Platform (MHP) standard. MHP defines a Java based execution platform for interactive applications. Next to a subset of standard Java APIs it defines a number of APIs to control the digital receiver features of a set-top box, like tuner, service information database, and video decoder. MHP applications range from simple teletext like information services as a stock ticker to e-commerce solutions for home shopping.

Requirements

From the trends and features described above a number of new functional product requirements can be derived. Future products must support various forms of networking and connectivity. Furthermore, various mass storage media - both optical and magnetic - and file systems must be supported. But besides these new functions that seem to make consumer products more PC like, the traditional requirements from the high-volume consumer electronics domain still hold. People expect a high level of robustness and ease of use from TVs and DVDs. Frequent user reboots of a TV, because of system crashes, are no option. Given the huge product volumes and strong price erosion, the bill of material of mainstream consumer products is under constant pressure. This translates into a limited CPU cycle budget and a limited memory footprint that make efficient resource management very important. In combination with the required robustness and the real-time nature of audio and video, this leads to the need for efficient, deterministic, predictable CPU scheduling and memory allocation. Traditionally, small and efficient real-time kernels were the only solution for meeting these timeliness and predictability demands in a cost effective, resource constrained way. These small kernels like CMX, pSOS and VxWorks, however, do not offer the rich set of features that ”fat” operating systems like Windows and Linux do. Since future consumer products will depend more and more on a rich set of networking, connectivity, and storage features, the ”fat” operating systems are entering the consumer products market.

Technology

To what extend can Linux and other open-source software meet the requirements of today’s and tomorrow’s products? To answer this question we discuss three topics: real-time performance, memory footprint and functionality.

Real-time performance

The standard Linux kernel provides soft real-time support according to POSIX 1003.1b. A priority based preemptive scheduler is available, with scheduling latencies varying on a standard PC from a few μs to 100 ms or even more. Although average latency is usually very low, responses in the order of 10 ms occur frequently. Compared to for instance 20 ms deadlines for a 50 Hz. video frame rate, it is clear that - unless a lot of expensive buffering is applied - plain Linux cannot cope reliably with the requirements from the video domain. This does not mean that you cannot play DVD movies on a standard Linux PC. Such a standard PC uses several Mbytes for buffering audio and video. And even then, when starting a web browser or receiving an e-mail while playing the movie, on many occasions the video is not displayed smoothly at a constant frame rate. Therefore, from a real-time requirements perspective, the only way to build a robust consumer device based on a standard Linux distribution is by solving all real-time requirements with extra hardware. This can either be more memory for buffering, but a more reliable strategy is adding an extra processor that takes care of the real-time control tasks. In such a dual processor system the processor running Linux can be seen as an application co-processor for performing the non real-time best effort application tasks.

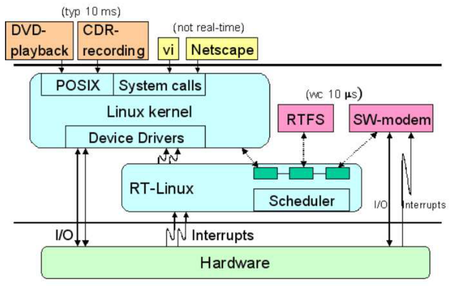


Figure 2: Real-time Linux, a hybrid solution.

But Linux is open-source. So for each problem or lacking feature somewhere in the world someone can be found working on it. Although this is not true for all lacking features, it does hold for the real-time performance of Linux. And - following another open-source tradition, the Darwinian survival of the fittest - it is not a single person or project that is providing a solution for improving the real-time performance of Linux. A number of (sometimes competing) alternative solutions exist. These solutions can be classified in two groups, either enhancing the real-time performance of the Linux kernel itself or enhancing system performance by combining Linux with a small real-time kernel.

Enhanced Linux

The greater part of the large worst-case scheduling latency of standard Linux is caused by its monolithic kernel design. Although user space activities are scheduled preemptible, kernel space activities are not. This means that operations like system calls, ”bottom half” interrupt handlers and process scheduling are completed once started, even if an other higher priority activity is triggered (e.g. by an external interrupt) and ready to be executed. To reduce preemption delay in kernel space two solutions are being worked on. The first one, called low-latency patch, shortens the non-preemptible kernel paths by introducing explicit reschedule points inside the kernel. The second one, called preemption patch, exploits kernel extensions for supporting symmetric multi-processor architectures (spinlocks). Instead of allowing preemption at specific points inside the kernel, like the low-latency patch does, the preemption patch enables preemption for the complete kernel space, except for specific critical sections that should not be entered by more than one thread concurrently. Most of these critical sections are already protected by spinlocks for the multi-processor version of the kernel. By implementing the spinlocks with mutexes in the single processor build - instead of skipping the spinlocks like the standard kernel does - the kernel becomes re-entrant. Of course the complete solution is somewhat more complex than described here, but a detailed discussion of both patches is not the goal of this article. Although it is difficult to give accurate worst case scheduling latency numbers for both patches - numbers depend on different versions of the patches, different hardware and different benchmarks - it is generally assumed that both patches can reduce maximal latency to sub ms numbers in the order of hundreds of μs.

Another approach to enhancing the real-time performance of Linux targets the scheduler. The standard Linux scheduler is optimized for throughput and fairness, not for real-time responsiveness and predictability. Although the scheduling policy for application processes can be set to the traditional priority based preemptive scheduling offered by real-time kernels, the implementation of the Linux scheduler still can be improved for better and deterministic performance. Other improvements to the scheduler include new scheduling policies like the budget based scheduling found in resource kernels.

Hybrid solutions

RTLinux and RTAI are two approaches to achieve hard real-time behavior, generally referred to as 'real-time Linux'. Essentially, these projects are constructing their own real-time kernel. This kernel runs Linux as its lowest priority task. All the standard services of the Linux kernel and Linux applications are available (although without the real-time guarantees), yet real-time threads and handlers can run with minimal, hardware limited, latencies. Real-time Linux solutions can guarantee interrupt latencies of several μs and scheduling latencies in the order of 10 μs The major drawback of these solutions is that the improved real-time performance only holds for the real-time kernel part. Standard Linux device drivers do not become real-time drivers by using a real-time Linux variant. Drivers may disable interrupts for up to hundreds of μs and violate other standard real-time design rules. In order to use those drivers in the real-time domain, they have to be rewritten.

Footprint

Although the memory available in consumer devices is growing, memory remains a scarce resource. The main reason is that the larger part of the Mbytes that are available in high-end digital products are used for buffering and rendering audio and video data. Also graphics consume more and more memory. Besides a framebuffer (sometimes doubly buffered) a considerable amount of memory is spent on built-in fonts and bitmap graphics. Finally, new features like a web browser or Java virtual machine have a large impact on the remainder of the memory that is not used for audio and video data. Traditional real-time kernel sizes are in the order of 100 Kbytes or 10 Kbytes for small kernels. A minimal desktop Linux system requires several Mbytes of memory, without a graphical user interface like X. Of course not all libraries and modules that are needed on a desktop PC are needed on a TV or set-top box. By selecting specific features and disabling others, standard Linux distributions are reasonably scalable. Some commercial embedded Linux distributors even provide special tooling for configuring and scaling down Linux. However, assuming Linux is chosen in favor of a real-time kernel because of its rich set of features, it is fair to state that a minimal embedded Linux solution requires several Mbytes of memory.

And that is just the kernel. When adding applications and services like a desktop window system, web browser and Java virtual machine, total memory requirements are in the order of tens of Mbytes..

But Linux is open-source. Just like on the real-time performance, several projects are working on reducing the memory footprint of Linux based systems. We name two examples here.

The standard GNU C library that most applications dynamically link to requires about 1.3 Mbytes. That's a lot of memory, especially if only a few functions are used - printf does not always make sense inside a TV. When linking statically to the library, code that is not needed can be stripped, resulting in a minimal size of about 300 Kbytes. But then, each application duplicates this code. Several small and scalable C library implementations, like 'uClibC', 'DietLibC', and 'newlib' have been or are being developed for embedded applications, reducing the memory requirements to several Kbytes.

Graphical user interfaces and window systems that are used in standard Linux versions do not meet the requirements of the embedded consumer domain. Besides a large footprint, standard solutions are not optimized for interlaced TV displays or small LCD screens. There are many open-source (and commercial) projects that deal with those issues. Again, we name two examples. The 'MicroWindows' project works on a small modern graphical windowing environment for embedded applications. It supports an API based on Win32 GDI, with a footprint below 100 Kbytes. 'Qt/Embedded' is a solution that is scalable from 800 Kbytes to 4 Mbytes. It aims, among others, at consumer devices like set-top boxes and PDAs.

Functionality

The technical motive for choosing Linux is not its real-time behavior or memory footprint. It is the rich set of features that ease development and enable a short time to market. Linux supports most of the new functions that we see entering consumer products in the coming years. It is a fully featured, high-end operating system that supports multiple processors, processes and users, that supports networking and many file systems, and that provides memory protection and security options. Apart from generic services and implementations of standard protocols, specific support can be found for a wide range of processors and peripherals, including standard (wireless) network cards and IDE drives, but also including a/v consumer domain specific peripherals like (digital) TV cards, IEEE 1394 high speed digital interface, and BlueTooth devices. Additionally, since sources of all drivers are open, the effort required to realize a new driver for not yet supported hardware is less than it would be when starting from scratch.

Next to the features that are built into the Linux kernel, many utilities, middleware and applications exist. Although most of them are targeted at desktop or server systems, a large number of them can be put to account for the consumer domain because of the convergence of PC and consumer functionality. Furthermore, because of the big momentum for embedded Linux solutions, more and more middleware and applications become available that are specifically targeted at embedded devices. One just has to take a quick look at a site like www.linuxdevices.com to realize this. We mention a few examples that are relevant for the consumer domain. Many projects deal with web browsing on embedded platforms. Both open-source solutions, like 'viewML', 'Konqueror/Embedded', and commercial solutions like the 'Opera' web browser that runs on top of the 'MicroWindows' environment are available. Other open-source projects and commercial companies are building (digital) TV receivers including electronic program guides and hard-disk video recording on top of Linux. Activities range from open-source initiatives like the 'vdr/LinuxTV' project - developing 'personal video recorder'[2] software -, via associations like the 'TV Linux Alliance' - standardizing platform interfaces -, to companies like 'TiVo', that offers a personal video recorder service and set-top box on subscription basis.

Figure 3: The TiVo personal video recorder.

[2]Personal video recorder is the phrase used for indicating hard-disk based video recorders that support functionality like a pause-button and automatic recording of all broadcasts of your favorite soap series.

Non-technical issues

How free is free software? Throughout this article, we deliberately use the term open-source instead of the term free software that is commonly adopted too. The free part of free software refers to the freedom towards users that is ensured by licenses. And although the business models associated with open-source software are usually not based on paying for the actual software and intellectual property that the software represents, free software is not free of cost. In order to use open-source software in software intense high-volume electronics products, for instance, professional support and training are important too. Today, the support you can get from companies specialized in embedded or real-time Linux is comparable to what you get from traditional real-time kernel vendors. Quality is good and prices are fair, among others because of the competition between different suppliers that do not hold technology locks on their customers with specific real-time solutions or tools.

Another cost factor that people tend to forget when talking about free software is licensing. Since there are several important licensing related issues, it deserves a section on its own.

Licenses

The idea of open-source software dates back to the software-sharing community of the MIT Artificial Intelligence Labs. When this community dissolved, one of these people, Richard Stallman, continued to write what he called free software. He later formed the Free Software Foundation that is responsible for many of the GNU applications. Several open-source licenses exist nowadays, like e.g. the GNU General Public License (GPL) and Library (or Lesser) General Public License (LPGL), the BSD or Berkeley license, and more recent variations such as the Netscape and Mozilla Public Licenses and the Sun Community Source license. The term 'copyleft' is also used, especially with the GNU licenses, because the central idea is to give everyone permission to run, copy, and modify the program, and to distribute modified versions. It is, however, not permitted to add restrictions. With many of the licenses modified versions of the software must, upon redistribution, provide the same freedom to users, including availability of the modified source code. One of the differences between these licenses is in the terms for combinations with proprietary software. For example, linking proprietary closed-source applications with GPLed libraries is not allowed, but linking them with LGPLed libraries is. Linking binary modules into the Linux kernel is generally allowed due to an explicit license exemption granted by Linus Torvalds.

The availability of modified source code that GPL and GPL-like licenses require, has both advantages and drawbacks. It enables open-source communities and projects and has helped Linux to become the feature rich operation system that it is today. For high-volume consumer electronics manufacturers it means lower bill of material costs, since no run-time license fees are due. But it can also mean that the manufacturer's specific intellectual software property and added value is no longer protected when specific enhancements and additions to the copyleft licensed source code have to be opened up. And when this intellectual property is also protected by patents, the situation becomes very unclear from a legal point of view. By incorporating GPL licensed software into a product, the manufacturer grants users the right to freely use and distribute the manufacturer's modifications and additions. On the other hand, patent laws disallow the free use of these modifications and additions when they are protected by patents.

Because of the fact that open-source licenses have never been tested in court, companies that consider incorporating open-source licensed software into their product should make a trade-off between the time to market and cost advantages and the legal intellectual property risks. Next to this legal risk, also the public image of the company should be considered. In the spirit of freedom of use and sharing, the open-source community has little respect for companies that only use software for their own commercial benefit. Furthermore, the principles and ideas behind open-source software are not in line with traditional patent protection. On the one hand many members of open-source communities feel that software patents, that restrict the free use of software, hinder innovation. On the other hand, patent-minded people feel that the protection and possible financial rewards stimulate companies to invest in innovation. Without proper protection of intellectual property rights, the huge research and development investments that are needed for breakthrough innovations would not be affordable.

To overcome some of the disadvantages of GPL, companies like Sun have introduced semi-open-source licenses to exploit the open-source advantages while simultaneously protecting their intellectual property and taking benefit from community-constructed add-ons. The dual licensing of Qt's graphical windowing environment is another approach. It can either be used free of charge, but protected by GPL, or companies can choose to commercially license the same software that can be combined with own applications without restrictions.

Process

The final subject we discuss does not concern the products of open-source, but the way these products are developed. Given the fact that traditional development processes do not scale up very well to cope with the large team sizes that are needed for complex products, open-source development processes can be an interesting alternative.

Re-use of software is often seen as the solution for coping with the increasing complexity and strong time to market requirements. Software re-use, however, turns out to be very challenging in practice. It is very difficult to take an arbitrary software component from one product and put it to use in another product. Reasons vary from strong context dependencies of a component, quality problems and lacking documentation, to architectural mismatch. One way of eliminating these obstructions for re-use, is to centrally enforce a common architecture and common processes for documentation, quality control, etc. However, when development projects grow in size - a modern, high end TV requires over 100 man years of software - or are executed over different locations and time zones, the overhead of centrally enforcing and checking the architectural and process rules grows exponentially, if possible at all.

The other way of dealing with the re-use obstacles is the open-source way. The distributed nature of open-source projects, with many contributors that communicate through the Internet, scales much better than a centralized approach. The basic idea is very simple. When programmers can read, redistribute, and modify the source code for a piece of software, it evolves. People improve the code, adapt it, and fix bugs. And this can happen at a speed that, in comparison to the pace of conventional software development, may seem astonishing. Several companies are experimenting with open-source like development processes. Either through projects that are truly open to the public community, or through so-called 'inner-source' projects that aim at creating and leveraging communities inside a company. Main challenge for these experimental development processes and organizations is to find the right balance between centrally managed and distributed development activities. Without some central guidance and direction, software will not evolve into the right (commercial) directions.

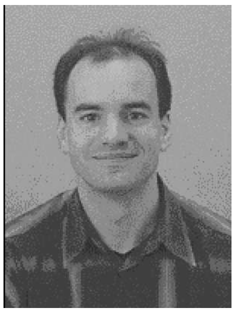
Conclusions

Linux is gaining a lot of momentum in the consumer electronics domain. Although real-time behavior and memory footprint do not yet allow Linux to be used in all mainstream consumer products, the rich featuring makes Linux a serious candidate for today's and a very serious candidate for tomorrow's high-end products.

Using Linux and other open-source software can be very tempting from a technical and time-to-market point of view. The freedom that open-source advocates, however, is mainly freedom to end-users and does not necessarily match with the intellectual property business interest of consumer electronics manufacturers. Given the legal uncertainty - open-source licenses have never been tested in court - attention must be paid to reducing the risks, for instance by avoiding linking to GPL software or only linking binary modules into the Linux kernel.

Open-source software influences the consumer products of tomorrow. If not by being incorporated into products, then by adopting certain aspects of the open-source development process that promotes sharing and re-use of software. This leads us to our final conclusion. Linux inside your TV? Probably sooner than you think!

Ruud Derwig (Ruud.Derwig@philips.com) is working at Philips Research on software platform architectures for resource constrained products in the consumer electronics do-



main. Key areas of expertise are real-time kernels and operating systems, resource aware component architectures, and heterogeneous software architectures. Before joining Philips Research he followed the post-masters Software Technology program (OOTI) at the Eindhoven University of Technology.

Recent OOTI Publications

The post-masters programme OOTI is concluded with a design project. The final reports of these projects are in general publicly available, unless stated otherwise. The following reports have been published lately.

M. Hudak *Internet Tuner*,
Keywords: Internet tuner/Broadcasting/Video streaming
ISBN: 90-444-0168-8, 40p., December 2001

M. Kychma *Commercial Block Detection on Digital Recording Products (DVD and HDD)*,
Keywords: Commercial Block Detection/Audio/Video Retrieval/MPEG-2 Encoding
ISBN: 90-444-0192-0, 59 p., April 2002

S. Shumski *Scripting Interface Service*,
Keywords: MATLAB, Python, Script, IDL, Interface
ISBN: 90-444-0201-3, 37 p., April 2002

G. Muitjens *On the Suitability of Java for TV Control Applications*,
Keywords: Software components, Koala, TV Control software, Real Time Java, Java
ISBN: 90-444-0191-2, 65p., April 2002.